**USEFUL LINKS**

* Angular tutorial through w3schools - <https://www.w3schools.com/angular/>
* Angular directives reference - <https://www.w3schools.com/angular/angular_ref_directives.asp>
* Angular documentation and main site - <https://angularjs.org/>
* AngularJS services - <https://docs.angularjs.org/guide/services>
* Angular factory vs service vs provider - <https://tylermcginnis.com/angularjs-factory-vs-service-vs-provider/>

INSTALLING IN YOUR APP/PROJECT

**ABOUT ANGULAR**

* AngularJS is a JavaScript framework/library that can be added to an html page. It extends HTML attributes with “directives” and binds data to html with “expressions”.
* Angular models define angularjs apps, the controllers control it. The **ng-app** directive defines the application, the ng-controller directive defines the controller.
* Angular uses services, a service is a substitutable object that is wired together using dependency injection (DI), and is used to organize and share code across an app. For example, $http is a service. However, it is useful to create your own, so I most commonly do this with a factory, which is a service. A factory is also an object that you add properties to, then return it, and pass it to the controller.

FILE/FOLDER FORMAT

WORKFLOW

**THE BASICS**

* AngularJS extends HTML with **ng-directives**, some common and important ones are: **ng-app**, which goes at the top of index.html and defines an angularJS application, **ng-model** which binds the value of html controls (input, select, textarea) to application data, and **ng-bind** which binds application data to the HTML view. All directives start with “**ng-**”
* Angular expressions are written within html code in double curly brackets. Ex: if you have in your html page: **<h1>{{ 5+5 }}</h1>**, then 10 would be written as a header 1. But expressions can also be written inside a directive, ex: **ng-bind=“expression”**
* **JavaScript Expressions VS AngularJS Expressions:** Both can contain literals, operators, and variables. But AngularJS can’t be written inside HTML, nor can you use conditionsl, loops, and exceptions with angularjs.
* **ANGULARJS CONTROLLERS:** Are javascript objects created by a standard javascript object constructor. You define their application or scope with **ng-controller**, meaning that if I want a partial to use a certain controller, I would wrap the entire partial in say a div, and write **<div ng-controller= ‘controller\_name\_here’>**.

**THE DIFFERENCE BETWEEN USING AND NOT USING $SCOPE WITHIN THE CONTROLLERS**

* **$SCOPE:** is the application object or the owner of the application (controller’s) variables and functions. Basically, when used in a controller, it has access to and controls that controller’s variables and functions/methods.
* **HOW TO USE $SCOPE:** To use it, when creating a controller, you pass it to the controller function as a parameter, ex: **app.controller(‘NameHere’, function($scope){…..})**. Afterwards, any variables or functions in that controller need to have “$scope.” in front of them, or you won’t be able to access them in the view and possibly other parts of the controller, ex: $scope.name = “Jack”, $scope.test = function(..){…}. In the view, you don’t refer to these variables or methods with scope in front of them, for example, I would access the name variable above with **{{name}}**. Not sure how this would work if multiple controllers are used in the same partial and had a variable or method of the same name.
* **USING $SCOPE:** As shown above, you pass in “$scope” as a parameter to the controller function. Then, in order to use anything created in your controller in the view, you need to add “$scope.” to the front. A benefit to this is that you can reference that variable directly in the view, unlike when you’re not using “$scope” and you need to write the name or short name of the controller ahead of it (shown in the not using $scope bullet). See the controller example in green, and the partial in blue:

app.controller('$scope\_example\_controller', function(first\_factory, $scope){

$scope.name = 'Jack\_2\_ng-model\_example\_using\_scope'

$scope.arr = ['Hey', 'There', 4]

first\_factory.show(function(data){

$scope.users = data;

})

})

<div ng-controller='$scope\_example\_controller'>

<h1>All Users</h1>

<table>

<tr>

<th>First Name</th><th>Last Name</th><th>Fav Language</th><th>Actions</th>

</tr>

<tr ng-repeat='user in users'>

<td>{{user.first}}</td><td>{{user.last}}</td><td>{{user.fav}}</td>

</tr>

</table>

<form class="" action="index.html" method="post">

<input ng-model="name">

</form>

<h3>This partial and controller was made to demonstrate the difference between using $scope or not within the controller.</h3>

<p>How you show a variable with scope: The array 'arr' is: {{arr}}</p>

</div>

* **STILL USING $SCOPE:** As you can see, the benefit is referring the variables directly. See the next bullet to see the difference when you’re not using scope.
* **NOT USING $SCOPE:** Not using $scope focuses on using “this” and “self” instead. You don’t need to pass anything as a parameter to the controller function as a replacement for $scope. “**this**” is available in the controller function by default and refers to the controller and it’s variables, methods, etc.
* **STILL NOT USING $SCOPE, DIFFERENCE BETWEEN - SELF/THIS:** HOWEVER, an important note is how we use “**self**” in the controller. You can declare and access a variable in a controller like so “**this.name = ‘Jack’; console.log(this.name);”**, however, if you’re within a function that doesn’t belong to the controller, then the scope of “this” changes and refers to the object that owns the function. For example, see the controller below, and how I call the factories function “show” with “first\_factory.show(….)”. Inside the factory, the show function takes a callback function as a parameter and calls that callback function with an array of objects as a parameter. What’s important is that even though I write out what the callback function does with the array in the controller, it is called/activated within the factory, so when the function actually runs, it is within the factory. Therefore, if I write “this” in the function, “this” refers to the factory and not the controller. Notice the console.logs within the show’s callback function. doing “console.log(this.name)” shows undefined because there is no name variable in the factory. But “console.log(self.name)” shows the name variable from the controller. Basically, at the start of my controller I write “**var self = this;**”, this basically creates a variable and assigns the “this” that refers to the controller to it. You could name it anything, but another way to picture it is: “**var self = TheCurrentControllerWeAreIn;**”. To keep it simple, whenever we declare “self” like this, it means that no matter where we call it, it refers to the controller it was created in. See below, controller is green, comments are red, and the partial is blue:

app.controller('userListsController', function(first\_factory){

var self=this;

this.name = 'Jack\_1\_ng-model\_example\_not\_using\_scope'

first\_factory.show(function(data){

self.users = data;

console.log(this.name); //undefined, “this” refers to first\_factory

console.log(self.name); // 'Jack\_1\_ng-model\_example\_not\_using\_scope', no matter if we’re in the controller or first\_factory, “self” still refers to the controller.

})

})

<div ng-controller='userListsController as user\_controller'>

<h1>All Users</h1>

<table>

<tr>

<th>First Name</th><th>Last Name</th><th>Fav Language</th><th>Actions</th>

</tr>

<tr ng-repeat='user in user\_controller.users'>

<td>{{user.first}}</td><td>{{user.last}}</td><td>{{user.fav}}</td>

</tr>

</table>

<input ng-model="user\_controller.name">

<input type="submit" name="" value="Test" ng-click="user\_controller.test()">

</div>

* **STILL NOT USING $SCOPE:** The other important part about not using $scope is how you do things in the view/partial. Basically, we have to reference all of our variables with “**controllerName.**”, so since our controller in this example is named “userListsController”, if we wanted to show the “name” variable, we would have to write “{{**userListsController.name}}**”. This is a little long and annoying to type, so we can create a sort of nickname/short name to use instead. Whenever you write “ng-controller=”, followed by the name of the controller, we can write “as nickname” after it. This creates another name that we can use to reference that controller and it’s variables/methods. For example, if I write “**<div ng-contorller= ‘userListsController as users’>**”, then I could access the name with “**{{users.name}}**”, much easier to write.

**ANGULARJS DIRECTIVE BASICS AND EXAMPLES**

* **NG-APP:** is where you define the root element of the angularjs application. This directive will “auto-bootstrap” (automatically initialize) the application when a web page is loaded. This can go at the start of your body but is better off at the start of your index.html file, ex: **<html ng-app='myApp'>**
* **NG-INIT:**is where you can basically declare variables or gives things values. For example, if I wrote **<div ng-init=“myCol= ‘lightblue’ “>**, then in another line wrote **<input style= “background-color: {{myCol}}” ….>**, then it would read that as “background-color: ‘lightblue’”. Another example, you could declare two variables by doing **<div ng-init= “quantity=1; cost=5; name=‘John’”>**, and use either variable as you want. Every directive, ng-init included, is like class in the sense that you put them in tags, ex: <div directive\_here>, <input directive\_here>, <p directive\_here>. You could also create objects or arrays using ng-init.
* **NG-BIND:** like it sounds, literally binds a value to something. If I wrote: **<p> The third result is <span ng-bind=“16”></span></p>**, then it would translate to “The third result is 16”.
* **NG-MODEL:** This directive allows you to bind the value of an input field to a variable created in angularJS, for example, if in my controller I have **this.name = ‘Jack’** (not using $scope in this example), and then had an input like this: **Name: <input ng-model=“controllerName.name”>**, then this would show up on the page as an input with “Jack” written in it to start. This is a two way binding however, meaning that if I click that input and typed to change it to “Jack swanson”, then the value of “this.name” in the controller would also change, until I reloaded the page/partial, they are linked.
* **NG-REPEAT:** is a directive used to repeat an html element, similar to a for loop in javascript. For example, if I wrote **<div ng-init= “names=[‘Jani’, ‘Hege’, ‘Kai’]”>**, then wrote:

**<ul>**

**<li ng-repeat=“x in names”>**

**{{x}}**

**</li>**

**</ul>**

* **STILL NG-REPEAT:** Then it would show each index in the names array as a list item. You could also use this on an array of objects (like data you get from mongoDB sometimes). For example, if names was **[ {name: ‘Jani’, country: ‘Norway’}, {name: ‘Hege’, country: ‘Sweden’}, {name: ‘Kai’, country: ‘Denmark’} ]**. Then you would write the repeat code as:

**<ul>**

**<li ng-repeat=“x in names”>**

**{{ x.name + ‘, ’ + x.country }}**

**</li>**

**</ul>**

* **CREATING DIRECTIVES**: I have yet find a need to create a new directive but you can with the **.directive** function, see how in this url: <https://www.w3schools.com/angular/angular_directives.asp>, in the middle of the page is where it shows it.